Defect Prevention and Detection in Software for Automated Test Equipment

Master's Project Report

Eric Bean
ebean@byu.net

Rev: 11/28/2006

This project report is submitted to the Department of Electrical Engineering and Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science

Abstract

Software for automated test equipment can be tedious and monotonous making it just as error-prone as other software. Active defect prevention and detection are also important for test applications. Incomplete or unclear requirements, a cryptic syntax used for some test applications—especially script-based test sets, variability in syntax or structure, and changing requirements are among the problems encountered in one tester. Such problems are common to all software but can be particularly problematic in test equipment software intended to test another product. Each of these issues increases the probability of error injection during test application development. This report describes a test application development tool designed to address these issues and others for a particular piece of test equipment. By addressing these problems in the development environment, the tool has powerful built-in defect prevention and detection capabilities. Regular expressions are widely used in the development tool as a means of formally defining test equipment requirements for the test application and verifying conformance to those requirements. A novel means of using regular expressions to perform range checking was developed. A reduction in rework and increased productivity are the results. These capabilities are described along with lessons learned and their applicability to other test equipment software. The test application development tool, or “application builder”, is known as the PT3800 AM Creation, Revision and Archiving Tool (PACRAT).

Keywords: Defect Prevention, Defect Detection, Mistake Proofing, Regular Expressions, Automated Test Equipment, Test Application Builder

Table of Contents

Table of Figures

Table of Tables

Acknowledgements

I would like to acknowledge the guidance of my fellow team members on the PT3800 project, Dave Irwin and Doug Gutekunst, for their mentoring and guidance.

I also acknowledge the guidance of Professor Hossein Saiedian in completing this project and throughout my graduate school experience.

Finally, and most importantly, I want express my sincere thanks to my wife, Kammi, and our two children, Kyle (3) and Bryce (1), for their patience and support.

1 Introduction

Automated test equipment plays an increasingly important role in the manufacturing world. This is especially true with the emphasis on lean manufacturing processes. Automated test equipment can be a cost effective means of testing products in the factory. Most automated test equipment is controlled by software. This software may be embedded, PC based, or even script-based.

Software development is a highly people-oriented activity and therefore, a highly error-prone activity. Defects may be injected during requirements analysis or specification of the test application, test application design, and test application development or coding. Defect removal would typically be done at review points along that process and during testing of the test application itself. Software for automated test equipment is subject to same defect injection issues as other software.

In particular, though, test equipment application development can be tedious and unvaried. Depending on the equipment and the product under test, it may also be complex. Often, it is domain specific. Typically, though, test applications are made up of similar types of functions involving some stimulus and a measurement. Test applications must meet both the requirements of the product to be tested and the automated test equipment. All of this is true whether it is embedded software or a script. These things make automated test equipment software as error prone as other software both in initial development and in maintenance.

As a result, active defect removal and prevention is required to ensure quality. The quality of the test equipment application may have a significant impact on the test equipment's success at detecting defects in the product under test.

A good software engineering process is designed to build quality into a software product during development and to remove defects as early as possible. Much of software engineering research is focused on defect prevention and removal. In most cases, the focus is on the software development process. Significant results can be made from following a well-designed, mature software engineering process. The key is following the process and continually improving the process.

Some work has been done in the area of automated software error detection tools such as Lint, though, such tools typically have limited error-checking ability or report false errors (Hallem, 2003). Integrated development environments have improved the software development experience and provide lots of tools to make software development easier. But for test equipment applications that are often scripts and sometimes written in proprietary languages with their own domain specific syntax, tools such as these are not as widely available.

This project report describes a test application development tool designed with a high degree of defect prevention and detection built-in. While this tool is specific to a particular tester, the PT3800, the approach that it uses may be employed for other automated test equipment. The PT3800 tester is a refurbishment of an older tester, the PT3300. This refurbishment provided an opportunity to improve not only the tester

but to improve the test application development experience. It was an opportunity to create a test application development tool known as the PT3800 AM* Creation, Revision and Archiving Tool (PACRAT). This paper details the built-in defect prevention and detection techniques employed by PACRAT.

1.1 Background

The PT3800 is a universal tester of basic electrical capabilities. Over 1000 products manufactured under a particular government contract have used its predecessor, the PT3300. About 1600 test applications were used with that tester. The PT3800 tester performs the following basic types of tests:

- **Continuity** – measures the voltage drop across two pins when a specified current is applied
- **Ohmmeter** – measures resistance between positive and negative leads
- **Voltage** – measures the voltage output pins when a voltage is applied between input pins; also measures the current through the negative input lead
- **Insulation Resistance** – measures the leakage current through a set of negative pins when a voltage is applied between a set of positive pins and a set of negative pins
- **Inductance and Capacitance Tests** – measures inductance and capacitance.

The test applications used by the PT3300 tester are text files defining the product to be tested, connections and adapters to the product under test, the tests to run with stimuli and limits to be applied, and the test points. Instructions to the tester operator regarding how to connect cable adapters to the product under test or to other adapters are also included in the test application file.

The new tester, the PT3800, has been designed to replace the old tester with the same functionality. Some enhancements have also been included. It is expected that all electrical products manufactured under the particular government contract will be tested at least once by the new tester before delivery to the customer. Approximately 100 new test applications will be needed per year for the foreseeable future. Therefore, there is continued need for maintenance of existing test applications and development of new test applications. The PT3800 test applications are required to contain all the same information as the PT3300 test applications and more, hence the need for a robust development tool.

1.2 Motivation for PACRAT

The PT3300 test applications are nearly free form text with only basic structure required. A significant amount of variation is allowed in the syntax. A cumbersome editor is used along with a rudimentary syntax checker. Application development is performed on an HP terminal.

The development of the PT3800 tester provided an opportunity to improve the test application format and the means of creating it. The following issues specifically needed to be addressed in the new PT3800 system:

* AM refers to “automated media,” specifically test application source code.

- **Incomplete requirements** – Not all syntax requirements for test applications were defined including the allowance of some undocumented commands. Incomplete or undocumented requirements are not uncommon problems in software development. This tool needed to have the test equipment's requirements for a test application built-in, or at least access to them. That meant that a mechanism for defining and capturing the requirements was required, as well.
- **Syntax and file structure variability** – The variability in syntax and file structure in test applications made the test applications more difficult to maintain. Furthermore, it led to mistakes that the syntax checker did not always detect.
- **Complex setup** – The setup portion of the test application was complex and tedious. A high degree of complexity or tediousness provides greater opportunity for defect injection. The new tool needed to simplify many of the most complex tasks.
- **Cryptic syntax** – A somewhat cryptic syntax was used in the old test applications. The new development tool needed to make the syntax less of an issue and generally provide an easy-to-use interface.
- **Syntax and data validation** – As mention above, the capabilities of the syntax checker on the PT3300 system were limited and only provided defect detection capabilities. The new application builder needed to have much more comprehensive syntax and data validation built-in. Further more, mistake proofing features were required.
- **Evolving requirements** – Because so many products are tested on the PT3300, the maintenance needs are high for its test applications. The same is true of the new PT3800 tester. The system needed to be flexible and expandable to meet these needs.

Ultimately, all these issues result in opportunities for defect injection leading to lower productivity. All are common problems of software, in general, and software for automated test equipment in particular. The PACRAT test application development tool addresses these issues by meeting several objectives surrounding defect prevention and defect detection.

2 Application Builder Objectives

The test application builder objectives regarding defect prevention and detection are described in the subsections below. The achievement of these objectives is described in greater detail in section 3.

2.1 Defect Prevention

One of Phil Crosby's absolutes of quality management is that "the system of quality is prevention" (Crosby, 1984). Crosby further stated, "the error that does not exist cannot be missed." He put forth the idea that all defects are caused and that anything that is caused can be prevented. In fact, one of the key process areas of level 5 "Optimizing" of the Software Capability Maturity Model is defect prevention (Paulk, 1993).

In the development of software, including software for automated test equipment, defect prevention may involve development process activities such as the following:

- Planning defect prevention activities
- Tracking defects
- Identifying the root causes of defects
- Systematically eliminating causes of defects
- Joint Application Development with the customer
- Use of formal methods for requirements analysis and design
- Structured coding methods
- Formal test case construction
- Peer reviews throughout the development process (since developers learn to avoid the kind of defects that reviews detect)
- Continuous process improvement

Each of these process activities is important. However, this project sought to go a step further by building into the development tool defect prevention and mistake proofing features based on many of these process practices while addressing the issues experienced previously with test application development on the PT3300. The test application builder had the following objectives for defect prevention:

- Build tester requirements into the tool through the use of an application database
- Simplify and automate complex tasks
- Reduce or eliminate syntax variability
- Eliminate file structure variability
- Encourage the use of the development process

2.2 Defect Detection

Studies have shown that defect correction is less costly when the defect is detected early in the process. The longer a defect is allowed to ripple down through a process the more costly it is to find and correct. Defect detection and removal practices usually involve one or more of the following:

- Peer reviews or inspections
- Various types of software testing (unit, integration, acceptance, etc.)
- Automated error detection tools

The goal with this test application builder was to sharply reduce the number of defect escapes during test application development. This meant building active defect detection into the application builder. This was to be done in two primary ways:

- Data entry validation against the tester's requirements
- Source code analysis via comprehensive file validation against the tester's requirements before test application is used

2.3 Other Objectives

Other objectives of the test application builder included:

- Providing for future expansion due to changing requirements
- Improved maintainability of test application source code
- Reduced test application development time and therefore, increased productivity

These objectives further address the issues experienced with the PT3300 test application development.

3 Achievement of Objectives

3.1 Defect Prevention Objectives

In order to be effective at building defect prevention into this test application builder, it was necessary to understand where the potential defect injection points were in the test application development process. The experience of test application engineers who had used the old system was valuable in identifying problem areas. Furthermore, a Six Sigma Failure Mode and Effects Analysis of the development process was performed. Thus, the root causes of potential defects in a test application were identified and means were added to the application builder to prevent them. The result was the achievement of the objectives described in subsequent subsections.

3.1.1 Build tester requirements into tool through the use of an application database

Of course, thoroughly documenting the tester's requirements for the test applications is a significant step in dealing with incomplete or undocumented requirements. However, the tool needed access to the requirements so that the tool could ensure conformance to those requirements before a test application even got to the tester itself.

3.1.1.1 Formal requirements definition

The test applications for this particular tester are made up primarily of commands, associated parameters, and test limits in some sequence based on defined test options. A database was developed defining test application commands and their parameters, and the rules regarding their usage, syntax, and appropriate values. The database serves as a repository of the PT3800 tester's requirements for test applications. This database is used by PACRAT to ensure that a test application conforms to these requirements. Specific details about the tables and their fields may be found in Appendix B.2.

The database defines the commands available for the tester and the rules governing their use. Formal definition of rules regarding the occurrence and location of commands in the test application is allowed for in the database. A special syntax was developed for identifying rules regarding command order and location within a given section of a test application.

This database contains parameter definitions for each parameter set used by the commands. More than one command may reference the same parameter set. These parameter definitions include rules regarding the occurrence of a given parameter, whether it may recur in a command or if it is an optional parameter. In addition,

validation rules are included for most parameters along with default values. Approximately 93% of test application parameters in the database have some rule for validating format, syntax, and/or value.

Other requirements imposed by the tester are included in the database such as the valid ranges for voltage and current for certain types of test commands. In this way, requirements related to the power capabilities and limitations of the test equipment are included in the database. Damage to the tester itself or the product under test is prevented by verifying voltage and current are within the valid ranges. Other requirements in the database include contact patterns for adapter cables and valid user preference items and their validation rules.

The application builder, PACRAT, reads these rules from the database and uses them to ensure that the test application being developed conforms to the requirements of the test equipment. As much as possible, test equipment requirements for test applications were placed in the database. Some requirements, however, could not be included in the database. These were built into the development tool.

3.1.1.2 Requirements used to restrict choices

One way to prevent defects in any product, software or otherwise, is for the tools, templates, and jigs used to disallow incorrect placement of parts or connectors. In the software development process this may occur via the IDE used or the compiler used. For example, an object must be defined before it is used.

Only valid commands are allowed in PT3800 test applications. Furthermore, certain commands must occur in certain sections and in a certain order within the test application. PACRAT only allows developers to add commands to certain sections. Others are pre-populated with valid commands. Furthermore, when a command is added, the choices for the type of command are restricted to only valid choices using a drop-down list box. Valid choices are based on the location in the test application.

In addition, only valid parameters are allowed for a given command in PT3800 test applications. Furthermore, parameters must often occur in a certain order. Also, some parameters are required and others are not. Additionally, some parameters or parameter groups may be recurring within a given command. PACRAT only allows valid parameters to be added to a command. In those cases where parameter order is important, PACRAT ensures that parameters are added in the right location. Valid choices are based on the location in the test application.

Just as the choices of commands and parameters and their locations are restricted to only valid choices, where possible, values for certain parameters are restricted to only valid choices. In software development, the IDE may also restrict choices. For example, the IntelliSense features of Microsoft's Visual Studio encourage the selection of only valid methods or properties of an object.

Regular expressions and other special syntax are used extensively to define these requirements. As mentioned above, the use of formal methods is a software development process tool that aids in defect prevention. The use of regular expressions and other syntax provides a formal definition of test application

requirements. These requirements are then a part of the tool and applied every time a test application is created or edited. The use of regular expressions will be discussed further in section 3.2.1.1.

With the PT3800, some command parameters have a discrete set of allowed values as defined in the application data database. Regular expressions also are used to define these lists of discrete choices. PACRAT uses a drop-down list or other means of restricting entered values to only valid choices when editing those parameters that have a discrete set of allowed values. If a regular expression is used in the form "^(Choice1|Choice2|Choice3|etc.)$" with no special characters (?, parentheses, etc.) in the choice name then PACRAT detects the alternation and will use that to generate a drop down list of the valid choices when editing the parameter. For example, the Keyword parameter of the Voltage test command has the following regular expression for its validation rule:

```
^(OFFSET|DEVIATION|CVR|ON|ZENER|PAUSE|RATIO)$
```

PACRAT detects the pattern of discrete choices in the regular expression and sets up a drop down list box with the choices limited to OFFSET, DEVIATION, CVR, ON, ZENER, PAUSE, and RATIO. Approximately 52% of test application parameters restrict parameter values to valid choices based on the parameter's validation rule.

This database has proven to be an invaluable tool especially when dealing with new or changing tester requirements. Often changes can simply be made to the database without changes to the PACRAT source code.

3.1.2 Simplify and automate complex tasks

The complexity level of software source code has an impact on the defect density of that code. Figure 1 below illustrates this relationship. The bathtub shape of the curve indicates that the simple source code and highly complex source code both tend to have higher defect densities (Laird, 2005). In some ways, this may be counter-intuitive.

The simple source code is subject to defects because of its simplicity. Errors due to copying and pasting code and not appropriately adapting it to the circumstance would fit in this category. While some source code may be simple, it may also be tedious or monotonous. Developers may become complacent when writing such code. Much of the software for automated test equipment would fit in this category, particularly when dealing with scripts.

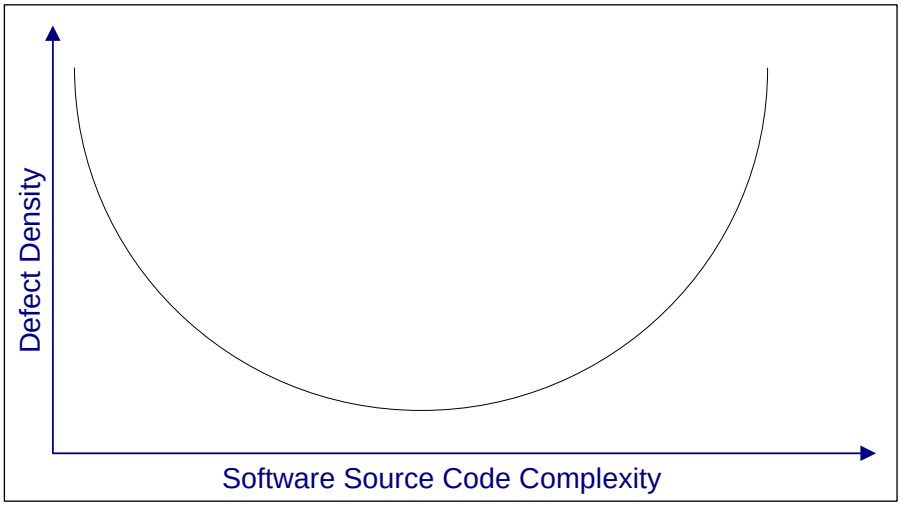

Figure 1 Software Complexity vs. Defect Density

The higher defect density for more complex software is more intuitive. The more complexity involved, the more there is to keep track of and the more opportunity there is to inject errors. Portions of test equipment software may have some complex setup or test requirements and is subject to the higher defect injection rates. The goal then with test equipment software is to reduce the tediousness of writing the simpler portions of the test application and to reduce the complexity of the most complicated portions of the test application.

The application builder developed for this project eased the more simple, yet tedious and boring tasks in the test application development process by automating those tasks, where possible, and making them even easier in other cases. Data validation, discussed in section 3.2.1, provides additional assurance against defects resulting from the tedious entry of data for setup and definition tasks in the test application. Automating or simplifying such tasks not only helps to reduce the time it takes to do them but reduces the risk of defect injection.

One of the reasons PACRAT has been able to successfully reduce the test application creation time (see section 3.3.3 below) is that PACRAT has significantly eased the definition and setup tasks that are a part of test application development. This is done in a variety of ways.

PACRAT allows test applications to be created, manipulated, and validated without requiring the developer to be aware of the file structure. Test application developers are free to focus on the testing approach required for a given product and not on file structure and syntax. In fact, developers are completely unaware of the file structure. PACRAT has been designed to be easy to use. The nature of the PT3300 test application file format increased the risk of defect injection. This risk is actually magnified with the XML format of the PT3800 system, but the use of PACRAT completely mitigates this risk.

PT3800 test applications all have to define the application itself and the product to be tested. These tasks are simplified through an easy-to-use graphical user interface,

pre-populated default values read from the application database, and through the automated generation of some required parameters based on the database. The application builder also dramatically reduced the complexity of the most complicated and confusing tasks in the old test system such as definition of cable adapters and their mapping to product connectors. Some specific features of PACRAT designed to simplify and ease the setup and definition of a test application and the tests contained in it are described in the subsections below.

3.1.2.1 New Test Application Creation

In addition, PACRAT has provided some utilities for creating a new test application that ease definition and setup tasks. These utilities are designed to reduce the risk of defect injection during the initial setup and definition of the test application.

The process of creating a new test application file can be tedious. The initial setup of the test is particularly dull. At new file creation, the minimum required sections, commands, and parameters are automatically generated. Of course, these need to be edited by the test application developer but the framework of a test application is ready at the start.

Going a step further, the New File Wizard feature does all that and provides the developer with the means to pre-populate a new test application with additional commands for defining product connectors, cable adapters, hookup commands, and test and connection verification commands. As with the New File feature, the developer must edit the parameters of these commands but a large framework for the test application may be easily put in place before editing any of the commands. In this way, the monotonous task of simply adding commands is eased.

As with other software, test equipment applications, or portions of them, are often similar to previously written applications. Developers often make use of the copy and paste features of their editor in such circumstances. This is often an error-prone practice since required customization or adaptation of the source code is easily neglected. To facilitate reuse and to prevent common “copy and paste” errors, PACRAT provides a means to “clone” an existing test application. The cloning process allows the developer to select which sections to copy from an existing file when creating a new test application file. Obviously, this is useful if there are many similarities between products to be tested, between the tests to be performed for two products, or between the cable adapters to be used in testing two products.

3.1.2.2 Defining Product Connectors and Cable Adapters

Among the most tedious setup and definition tasks in the old PT3300 system was defining cable adapters to be used. Its implicit mapping between adapter cable and contacts on product connectors lent itself to confusion and error. PACRAT dramatically improved the process for defining product connectors and their associated cable adapters. Many of the parameters for defining a product connector are automatically generated with the entry of some basic information and the click of a button. Likewise, the definition of cable adapters primarily involves dragging and dropping product connectors onto a grid representing the adapter. Again, by simplifying a process and/or interface and restricting choices to valid data, the defect injection risk is significantly reduced and mistakes are prevented.

3.1.2.3 User-defined Limit and Setup Definitions

PACRAT provides a means to create user-defined setup and limit definitions that are stored in the test application file and may be applied to test commands when they are edited. Setup definitions allow the developer to create user-defined setups for test commands. In this way, common values for various parameters may be set in advance. When a command is created or edited, the developer may select the desired setup. The developer may modify any of the parameters after selecting the setup. Often, the same test setup is used for several instances of a particular command with only the test points changing.

Limit definitions allow the developer to create user-defined limits that may be used for test commands requiring limits. When a command is created or edited, the developer may select the desired setup. Limit information is automatically entered when a user-defined limit definition is selected. Likewise, it may be modified at that time by the developer.

These features provide a convenience to the test application developer. But, they also prevent defects that may occur from having to type the same or similar information over and over by reducing the tediousness of the task.

3.1.3 Reduce or eliminate syntax variability

The syntax of each command and each parameter is explicitly defined in the aforementioned database. The syntax rules were made as restrictive as possible while allowing for reasonable amount of flexibility. In most cases, regular expressions define the syntax. The regular expressions provide a formal definition of the syntax requirements. See section 3.2.1.1 below for more details about how regular expressions are used in the application builder.

3.1.4 Eliminate file structure variability

A consistent, XML-style file structure was implemented by PACRAT. This file format actually made the complexity of the file structure worse than the previous format, but it made the structure consistent. This and other advantages outweighed the added complexity. Furthermore, because PACRAT removes the test application developer from the details of the file structure entirely, there is no room for variation in the format or structure of the test application. Every time a test application is saved in PACRAT it is always saved in the same structure regardless of the product to be tested or the personal style of the developer writing the tests. Maintenance of test applications becomes much simpler. Maintainability is an important quality factor for any software application.

3.1.5 Encourage the use of the development process

With all software development, the process used has a significant impact on the quality of the output. The test application development process is no different. PACRAT enforces certain aspects of the AM creation process to ensure that only a quality test application is used for product acceptance testing.

The PT3800 requires that test applications be in a “released” state before being used for acceptance testing. The released file is required to be validated and checked out on

the tester before being released to catch any final defects before being used to test product for acceptance testing. A released file may not be edited without changing the issue and configuration control suffix. A development test application must never be used for acceptance testing.

PACRAT does not allow a PT3800 test application file to be released unless it has previously had a status of “VAL,” implying that it has been “validated” both in PACRAT and at the tester. The status will not be changed in PACRAT unless the application has passed an automated comprehensive file validation (discussed later) to check for file structure and syntax errors and data errors. If file validation passes the status is set to “REL” and the file is made read only. PACRAT does not allow a released file to be edited. It will allow for the “next” suffix and issue to be created and edited. Development test applications are never allowed to be released for acceptance testing. In addition, a checksum is included in the file by PACRAT making it possible, in most cases, for the tester itself to detect if the file was edited outside of the tool and thus, was not subject to the defect prevention and detection features of PACRAT.

In addition, the Open Next feature is provided by PACRAT to provide a simple means of incrementing the configuration control suffix and issue of a released test application file. This is designed to help the test application developer follow the established process for making changes to a released test application.

These features to encourage or enforce the use of the established development process are not intended to create added hassle for the test equipment engineer. The intent is not to prevent the developer from doing reasonable things. Rather, the intent is to lead the developer through the process to the extent possible.

3.2 Defect Detection Objectives

Active detection and removal of defects is critical to the quality of a software product. The same is true of software applications for automated test equipment. A common development practice is the peer review. Peer reviews or inspections are one of the most effective defect detection practices available. The repetitive and tedious nature of test equipment software often creates and environment where the developer(s) may be complacent in a review. Special care is required to make such reviews effective for test equipment software (Barela, 2005).

To augment other defect detection practices in the software development process, significant automatic defect detection features were built into PACRAT. While some automatic repair or correction of test applications would be possible, a design decision was made to require the test equipment engineer developing the test software to make those corrections. However, resolution of validation failures is required before moving on in the development process. The primary mechanisms for defect detection employed by PACRAT are described in the subsections below.

3.2.1 Test Application Data Validation

PACRAT employs many methods to ensure that a test application is correct structurally and syntactically. Furthermore, PACRAT validates the data values, where possible, to ensure that a test application does attempt to exceed the capabilities of the PT3800 tester. These methods are described below along with the role of regular

expressions in data validation. These methods involve defect detection, thereby reducing escapes during test application development.

The most common form of data validation in PACRAT occurs when a command is being edited. In many cases, after a parameter value is entered, the user gets visual feedback if the value does not conform to the requirements for that parameter. Additional validation occurs when a command is saved. Any requirements regarding interaction among parameters are validated at this time. The presence of required parameters is also verified. Limit data is also validated, if necessary. All the above mentioned validation must "pass" in order for the command to be saved.

The parameter validation described above is both a means of defect *prevention* and a means of defect *detection*. It may be considered prevention because the validation happens so early in the process, even as parameter data is entered and before a command may be saved. In this way, the early defect detection prevents many errors from being saved at all.

3.2.1.1 Regular Expressions

The use of regular expressions allows for easy parsing of strings and more importantly, easy validation of syntax and data values. To a large extent, regular expressions also allow for validation of correct data values. Defining regular expressions for command parameters provides a means of formally defining the requirements for these parameters.

In the PACRAT application data database, most parameters have a regular expression associated with them defining valid syntax and values. By validating syntax, correct structure is ensured and mistakes in syntax are disallowed or prevented. Some examples of regular expressions for parameters are shown in Table 1. (The regular expression syntax conforms to the Visual Basic Regular Expression Engine 5.5 syntax.)

Parameter	Regular Expression Validation Rule	Examples of Valid Values
CurrentLimit	^(\d*\.?\d*)\x20+(u|m)?(Amp\x28s\x29)$	1.5 mAmp(s) .2 Amp(s)
DataSource	(?:([SRLZQABCDEPTV123456789])([h|t]))+?	ShLtEh3t AhEhPhTh
DrwgNum	^(A[MP]\-?[A-Z0-9]{6,7}(D\d\d)?)$	AM1A2345D01 AP-AB1234 AM1A77324

Table 1 Examples of Parameter Validation Rules using Regular Expressions

In addition, an algorithm for developing regular expressions capable of validating the valid range for numerical parameters was developed. Regular expressions are not typically used to validate that a numerical parameter lies within some expected range. The author has found no existing use of regular expressions in this way. However, since the mechanism for using regular expressions to validate parameters is already in the PACRAT software, it made sense to expand their use to include range checking where appropriate. This allowed for further use of the automation provided with regular expressions. Valid ranges for certain types of data were formally defined using

regular expressions. One advantage of doing range checking this way is that the same mechanism may be used for checking ranges whether the range is inclusive at either end or not. The algorithm for generating range checking regular expressions is described in further detail in APPENDIX A. An example of a simple range checking regular expression is shown below. The expression validates that the FontSize parameter in the DisplayMsg command is not less than 6 and not greater than 72.

```
^([6-9]|[1-6]\d|7[0-2])$
```

The extensive use of regular expressions had an impact on the test cases required to validate the PACRAT software (or, at least, the regular expressions themselves). For example, consider the case of performing range checking of a numerical value with a regular expression. Simple boundary value analysis is insufficient. In order to verify that the regular expression is correct, something akin to equivalence class partitioning is required. However, once a regular expression is verified to be correct, it provides a high degree of confidence that only valid data will be saved. A simple routine was written to test a regular expression. This routine was used to validate the correctness of a given regular expression by passing the expression and test values to the routine.

3.2.2 Test Application Source Code Analysis

As mentioned previously, the process of developing software is a highly error-prone activity due to the people-oriented nature of it. Methods of defect prevention and mistake proofing have been discussed. However, to ensure the quality of the resulting output of software development, active defect detection and removal is necessary. Source code analysis tools are designed to detect errors before run-time. A simulation of the program execution is a part of such analysis. With typical software, a thorough analysis of the source code and all possible execution paths is extremely difficult and may be nearly impossible. However, since many applications for automated test equipment are simpler and often script based, with a smaller command set, thorough source code analysis becomes more feasible.

PACRAT provides the capability to subject a test application file to a set of comprehensive file validation tests for the sole purpose of finding non-conformances (defects) to the requirements of the tester. The file is validated to make sure it has the required commands and required parameters for each command. Test IDs are validated to be unique. Loops are validated. Contacts used are validated. All PT3800 tester requirements for test applications that PACRAT is capable of verifying the test application against are validated. All possible paths through the test application are verified. This is possible since the application is basically sequential but with the possibility of up to 30 test options. A given command may be enabled for one or more test options. Where it is important, validation is done separately for the set of commands belonging to each defined test option. Specific details of the comprehensive test file validation can be found in Appendix B.4.

In addition, PACRAT requires a test application to pass file validation in order to be copied to the tester server. That means that a test application may not be loaded at the tester unless it has passed file validation in PACRAT. This is all designed to detect errors early in the process to prevent later rework on the test application itself and to prevent damage to the tester itself or to product under test, where possible.

3.3 Other Objectives

3.3.1 Maintainability

Maintenance of software may account for the majority of the total cost of development. In the old PT3300 system, the variability in syntax and file structure along made maintenance a more difficult task. This was further compounded by the incomplete nature of the documentation of test application requirements. Maintainability is improved in the new application builder because the requirements are "built-in" and the file structure is hidden from the developer. In addition, the easy-to-use nature of the application builder development tool improves the maintainability of test application source code.

3.3.2 Platform for Future Expansion

Two primary features of PACRAT enable it to provide a platform for future capabilities. The first is the object-oriented design which lends itself to easy enhancement and expansion. The second, and perhaps most important design feature, is the use of the application data database.

This database allows for expandability and adaptability for future needs without necessarily modifying the PACRAT source code. It has already proven itself to be an invaluable tool during PACRAT development when dealing with changing requirements. PACRAT was developed in an incremental fashion and is being used by the PT3800 tester development team in support of the tester development. As is the case with virtually all software projects, requirements change. In this case, frequently, a new command would be required or a parameter definition changed. Most often these changes simply involved adding to or modifying the data contained in the database. Source code changes for changing requirements were kept to a minimum and the tester developers did not necessarily have to wait for the next iteration of PACRAT to get the change they needed. It is important to note that such changing requirements are not just for PACRAT. These requirements changes are due to changing requirements of the tester itself. These changes may only impact test applications. Therefore, not only does this database of application data allow for the expansion of PACRAT, it allows for the expandability and adaptability of each test application developed using PACRAT. Maintainability is an important factor in software quality. So is flexibility.

3.3.3 Increased Productivity

One of Crosby's four absolutes of quality management is that "the measurement of quality is the price of nonconformance" to requirements. Furthermore, the second link in the Deming Chain Reaction following improved quality is lower costs. The third link is improved productivity. Clearly, one can expect a financial benefit to improved quality. It goes without saying that a reduced defect density in software has a direct relationship to the quality of the software. A significant side effect of active defect prevention is increased productivity. This is no different for the development of software for automated test equipment.

One of the key achievements of PACRAT is the reduction in the time it takes to develop a test application. Table 3 shows the approximate time it took to develop a test

application on the old PT3300 system based on the complexity of the test application. The equivalent time it takes to develop a similar test application using PACRAT is also shown. Data for hand editing the PT3800 test application file using a COTS XML editor is also shown. This method was used by test equipment engineers developing the PT3800 tester before PACRAT was sufficiently developed for use. The complexity level was determined by file size as defined in Table 2.

	Complexity Level		
	Simple	**Medium**	**Complex**
PT3300 Test Application File Size (in Bytes)	<150K	<400K	>=400K
PT3300 Test Application # of pages	2	6	10

Table 2 Test Application Complexity

	Test Application Complexity		
PT3300 Format	**Simple**	**Medium**	**Complex**
Test Application Development	4.0	8.0	16.0
Rework of test application files at tester	0.2	0.2	0.2
Verification of test application against product specification	0.3	0.6	1.0
Total Development Time	4.5	8.8	17.2
PT3800 Format With XML Editor			
Test Application Development	20.0	40.0	80.0
Rework at tester	0.5	0.5	0.5
Verification of test application against product specification	2.0	4.0	8.0
Total Development Time	22.5	44.5	88.5
PT3800 Format With PACRAT			
Test Application Development	2.0	4.0	8.0
Rework at tester	0.0	0.0	0.0
Verification of test application against product specification	0.25	0.5	0.75
Total Development Time	2.25	4.5	8.75

Table 3 Test Application Development Time (in Hours)

As can be seen from these tables, test application development with PACRAT takes less than half the time than development under the old PT3300 system. Development time with PACRAT is an order of magnitude less than hand editing the test application files with the XML editor. Of particular note is the reduction in rework. There is also a moderate reduction in the time it takes to verify a test application against the product specification. The reduced development time using PACRAT results in a significant cost savings as well.

4 Lessons Learned

The defect prevention and detection features provided by this test application development tool produce a demonstrable increase in productivity. That productivity increase results in a significant cost savings due to reduced development costs. The

reduction in defects at development time reduces rework and improves the quality of the test application.

While much of the implementation of this development tool is specific to the PT3800 tester system, a similar approach could be used in a development environment for other automated test equipment. Many of the principles applied in PACRAT could be applied in other tools.

4.1 Well-defined Requirements

Software for automated test equipment gets its requirements from two sources: the test equipment itself including any executive software running on it and the specification for the product to be tested. An ideal development environment ensures conformance to those requirements in order to guarantee quality in the test application source code.

In the case of the PT3800 tester, a large and varied assortment of products is tested, at least in part, on a single type of test equipment. In this case, the test application development tool could not feasibly incorporate the product requirements into the tool. Other test equipment that is more specialized to a specific product may be able to do so. In either case, the requirements of the test application are known.

The decision to use a database to capture tester requirements for the command set, parameter sets, and other requirements for test applications developed using PACRAT was probably the single best design decision made in the whole project. It truly proved invaluable. The database provided a means to clearly and formally define the requirements. Powerful defect prevention and detection capabilities in the development tool were possible because the tool could access those requirements to provide comprehensive verification of conformance to them.

Test equipment software typically is less complicated and involves a smaller command or function set than other types of software. Defining the functions or command set and as many of its requirements in a manner that a test application development tool can make use of those requirements will enhance the ability of any such tool to actively prevent and detect errors. It does not matter if a customized notation is developed or a standard notation is used. PACRAT made use of both a customized notation for command location and order requirements and regular expressions.

Regular expressions are often used for parsing text and for pattern matching. They are commonly used in internet applications requiring input from the user. They became a power tool in this project for formally defining many of the requirements for commands and their associated parameters. Regular expressions made the active validation of data as it was entered and as part of the comprehensive source code analysis possible. The novel use of regular expressions for range checking of certain parameters again provided a means of formally defining the range requirements. It also provided flexibility in the types of ranges (inclusive or non-inclusive) that could be verified using a single mechanism without hard coding anything about the ranges themselves. Using regular expressions for range checking is probably not the most efficient way to do range checking for most applications but it fit well in the case of PACRAT. A faster regular expression engine would also make the use of regular

expressions more useful in general. If the test equipment is to run in an embedded system, building the regular expression engine into the hardware may be an option to improve its speed (Lin, 2006).

4.2 Keep it Simple

The KISS* principle is in effect for test equipment software. Clearly, software complexity impacts software quality. While, test equipment software may be simpler, in some respects, than other software, it is not without some complexity. Any test equipment software development tool should seek to simplify the most complex parts of the application development. User-defined setups, macros, wizards, and other automated features make the application development simpler. Doing so not only reduces the time it takes to do perform those tasks, it also reduces the probability of defect injection during those tasks. Automating the most tedious and simple task also helps to prevent defects.

4.3 Reduce Variability

The goal of most quality programs such as Six Sigma is to reduce variability in a process whether in a manufacturing process or in a software development process. The resulting consistency leads to improved quality.

In the development of test application software, the reduction in the syntax variability and file structure variability is part of PACRAT's defect prevention scheme. Removing the details of the file structure from the application developer allows the developer to focus on the more important task of designing the tests to be performed. The same would be true of any test application development environment. This goes hand in hand with keeping it simple.

4.4 Encourage Use of Development Process

Even with all the built-in defect prevention in a development tool, following a good development process is an important part of ensuring quality. That is why PACRAT seeks to encourage the test application developer to follow the establish development process. It is a variation of the line from the movie *Field of Dreams*. Instead of "If you build it, they will come," it is, "If you built it in, they will follow." A test application development tool should encourage or enforce a good development process.

4.5 Verification of Conformance to Requirements

Validating conformance to requirements is critical to preventing escapes. The cost of defect removal increases as the latency of the defect increases. Detecting errors early in the development process prevents escapes downstream. PACRAT had a built-in means of detecting errors during development. Such a mechanism would prove useful in any application development environment.

Furthermore, the comprehensive test application source code analysis serves as a final check that the application conforms to the tester's requirements before it is executed by the tester executive. Again, having the requirements built-in or accessible by the development environment makes this possible. Full source code analysis is not always

* KISS – Keep it simple, stupid

feasible in most software applications. Such analysis is more likely to be feasible with test equipment software, especially script-based test software.

4.6 Maintainability and Flexibility

Easily maintained software is less likely to have additional defects injected as a result of a *bad fix* or an *enhancement*. All too often, with any software, the person maintaining it is not the person who originally developed it. PACRAT's ease of use and reduced variability make test application maintenance simpler. Such features could be built into any test application development environment.

Flexibility is another important quality factor. Evolving requirements is a universal issue for all software development. The software development process that handles changing requirements well is better equipped to ensure conformance to those requirements. For PACRAT, the application database provides a high degree of flexibility in dealing with changing or new requirements. Some built-in flexibility for dealing with requirements churn will aid the defect prevention and detection abilities of any test equipment development tool.

5 Conclusion

The PT3800 AM Creation, Revision, and Archiving Tool meets all of its objectives. Among its most valuable features is its ability to help the test engineer conform to the requirements of the PT3800 tester with regard to test applications. Conformance to the requirements of the product under test is up to the engineer. But, PACRAT provides a means to easily verify conformance to the product specification. As has been shown, PACRAT has significant capabilities to prevent defect insertion at test application development time. By itself, PACRAT may not guarantee zero defects in any given test application, but it serves as a highly-useful tool to help the developer achieve that goal. The result is an easy-to-use tool that saves development time. This in turn results in cost savings. The price of conformance is less than the cost of nonconformance to requirements. PACRAT is also highly valuable when it comes to defect detection. The early detection that the application builder provides will result in additional savings.

The database of application data proved to be invaluable in achieving the stated objectives for the test application builder. The database provided a powerful means of formally defining many of the test application requirements. Formal design methods are one means of preventing defects in software applications.

The extensive use of regular expressions also proved extremely useful. This includes the innovative use of regular expressions for range checking. Not only do they provide a means of validating test application data but they provide formal definitions of the requirements for a given piece of data. Future work would include enhancement of the algorithm to generate range checking regular expressions to handle decimal numbers. Furthermore, implementing the algorithm as a software utility would be beneficial.

6 References

Barela, S. (2005). “Software Review For Automatic Test Equipment.” *AUTOTESTCON*, IEEE.

Bowen, J. P. and M. G. Hinchey (2005). “Ten Commandments Revisited: A Ten-Year Perspective on The Industrial Application of Formal Methods.” *Proceedings of The 10th International Workshop on Formal Methods For Industrial Critical Systems*. Lisbon, Portugal, ACM Press.

Broberg, N., A. Farre, et al. (2004). “Regular expression patterns.” *Proceedings of The Ninth ACM SIGPLAN International Conference on Functional Programming*. Snow Bird, UT, USA, ACM Press.

Crosby, P. B. (1984). *Quality Without Tears: The Art of Hassle-free Management*. New York, McGraw-Hill.

Gutekunst, D. (2006). *Product Specification, PT3800, Issue B*. Kansas City, MO.

Hallem, S., D. Park, et al. (2003). "Uprooting Software Defects at the Source." *Queue* 1(8): 64-71.

Haruo, H. and P. Benjamin (2001). “Regular Expression Pattern Matching For XML.” *Proceedings of The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. London, United Kingdom, ACM Press.

Haruo, H., J, et al. (2005). “Regular Expression Types For XML.” *ACM Transactions on. Programming Languages and Systems* 27(1): 46-90.

Irwin, D. W. (2005). *PACRAT Specification, Issue 4*. Kansas City, MO.

Jones, C. (2004). “Software Project Management Practices: Failure vs. Success.” *Crosstalk: The Journal of Defense Software Engineering* 17(10): 5-9.

Laird, L. (2005). "In Praise of Defects." Stevens Institute of Technology. Retrieved 7 November 2006, from http://www.njspin.org/present/Linda%20Laird%20March%202005.pdf.

Liguori, F. (1971). “The Test Language Dilemma.” *Proceedings of The 1971 26th Annual Conference*, ACM Press.

Lin, C.-H., C.-T. Huang, et al. (2006). “Optimization of Regular Expression Pattern Matching Circuits on FPGA.” *Proceedings of The Conference on Design, Automation And Test In Europe: Designers' Forum*. Munich, Germany, European Design and Automation Association.

Linger, R. C. (1993). “Cleanroom Software Engineering For Zero-Defect Software.” *Proceedings of The 15th International Conference on Software Engineering*. Baltimore, Maryland, United States, IEEE Computer Society Press.

Martin, M., B. Livshits, et al. (2005). “Finding Application Errors And Security Flaws Using PQL: A Program Query Language.” *Proceedings of The 20th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, And Applications.* San Diego, CA, USA, ACM Press.

Oman, P. W. and C. R. Cook (1988). “A Paradigm For Programming Style Research.” *SIGPLAN Not.* 23(12): 69-78.

Paulk, M. C., C. V. Weber, et al. (1993). “Key Practices of the Capability Maturity Model, Version 1.1.” CMU/SEI-93-TR-025. Retrieved 2 May 2005, from http://www.sei.cmu.edu/pub/documents/93.reports/pdf/tr25.93.pdf.

Slaughter, S. A., D. E. Harter, et al. (1998). “Evaluating The Cost of Software Quality.” *Communications of the ACM* 41(8): 67-73.

Tomoko, M., M. Akito, et al. (2002). “A Method For Detecting Faulty Code Violating Implicit Coding Rules.” *Proceedings of the International Workshop on Principles of Software Evolution.* Orlando, Florida, ACM Press.

Troster, J., J. Henshaw, et al. (1993). “Filtering For Quality.” *Proceedings of The 1993 Conference of The Centre For Advanced Studies on Collaborative Research: Software Engineering - Volume 1.* Toronto, Ontario, Canada, IBM Press.

Wagner, S. and T. Seifert (2005). “Software Quality Economics For Defect-Detection Techniques Using Failure Prediction.” *Proceedings of The Third Workshop on Software Quality*. St. Louis, Missouri, ACM Press.

Zarrin, L. and P. Anne Banks (2005). “Quality, Cleanroom and Formal Methods.” *Proceedings of The Third Workshop on Software Quality*. St. Louis, Missouri, ACM Press.

APPENDIX A Range Checking Regular Expressions

A.1 Algorithm

An algorithm for generating a range checking regular expression was developed. This algorithm works only for whole numbers to validate a non-negative whole number *x* such that *min* <= *x* <= *max*. For this algorithm, the range must be inclusive at both ends. The main approach of the algorithm is to focus on the values allowed for each possible number of digits.

Where:
N = number of digits of the maximum
n = number of digits of the minimum
M = string representing the regular expression for the maximum
m = string representing the regular expression of the minimum
r = regular expression option element (i.e. the full regular expression is of the form $(r_i|r_{i+1}|r_{i+2}|\text{etc.})$
R = whole regular expression

Begin by generating a regular expression for the maximum value (R_M):

```
If N = n Then
     P = n
Else
     P = n + 1
End If
For i = N to P Step -1
     If i = N then
          For k = 1 to i
               Lmax = Mk – 1
               If k = 1 Then
                    rk = MiMi-1…[0-Mk]
               Else If k < i Then
                    rk = Mi…[0- Lmax]\d{k – 1}
               Else If k = i Then
                    rk = [1- Lmax]\d{k – 1}
               End if
               ri = ri & rk & |
          Next k
     Else
          If i > 1 then
               ri = ri & [1-9]\d{i – 1} & |
          Else
          ri = ri & \d
          End If
     End If
     RM = RM & ri
Next i
```

Now, generate a regular expression for the minimum up to the number of digits of the minimum:

```
For i = 1 to n
     If i = 1 Then
          r_i = m_n m_n-1 …[m_1..9]
     Else
          L_min = m_i + 1
          r_i = m_n …[L_min -9]\d{i – 1}
     End If
     R_m = R_m & r_i & |
Next i
```

If the number of digits for the maximum and the minimum are the same then consider the r_N in each. The value of L in each should be the same. If L = M_N Or L = m_n then throw out r_N from each. Otherwise, throw out r_N from R_M and change r_N in R_m to be

$$r_N = L\backslash d\{N - 1\}$$

Then merge the two regular expressions:

$$R = R_M \,\&\, R_m$$

Finally, throw away any duplicate choices in the resulting regular expression. The regular expression may be simplified, if desired. The expression should also be finalized by adding "^(" at the beginning and ")$" at the end.

A.2 Examples

Below are two examples using the above algorithm.

A.2.1 Example 1 (N = n)

Consider a whole number *x* such that 659 <= *x* <= 800. N = 3, n = 3, M = 800, m = 659.

Generate regular expression for the maximum (R_M) first.

```
For i = 3 to 3, k = 1 to 3:
     r1 = 80[0-0]
     r2: L = -1 so skip r2
     r3 = [1-7]\d{2}     :L = 7
```

$R_M = r_1 \,\&\, r_2 \,\&\, r_3 = 80[0\text{-}0]|[1\text{-}7]\backslash d\{2\}$

Generate the regular expression for the minimum (R_m).

```
For i = 1 to 3:
     r1 = 65[9-9]
```

```
r2 = 6[6-9]\d{1}    : L = 6
r3 = [7-9]\d{2}     : L = 7
```

$R_m = r_1 \& r_2 \& r_3 = 65[9-9]|6[6-9]\d\{1\}|[7-9]\d\{2\}$

Merge R_M and R_m by considering the r_N in each. The value of L in each should be the same. If $L = M_N$ Or $L = m_n$ then throw out r_N from each. Otherwise, throw out r_N from R_M and change r_N in R_m to be

$r_N = L\d\{N - 1\}$

In this case, L = 7 which is not equal to either M_3 or m_3, therefore:

$R_M = 80[0-0]$

And,

$R_m = 65[9-9]|6[6-9]\d\{1\}|7\d\{2\}$

And,

$R = R_M \& R_m = 80[0-0]|65[9-9]|6[6-9]\d\{1\}|7\d\{2\}$

which can be simplified and finalized to:

$R = ^(800|659|6[6-9]\d|7\d\{2\})\$$

A.2.2 Example 2 (N > n)

Consider a whole number *x* such that 43 <= *x* <= 598. N = 3, n = 2, M = 598, m = 43.

Generate the regular expression for the maximum (R_M) first.

```
For i = 3 to (2 + 1), k = 1 to 3:
     r1 = 59[0-8]
     r2 = 5[0-8]\d{1}
     r3 = [0-4]\d{2}
```

$r_3 = r_1 \& r_2 \& r_3 = 59[0-8]|5[0-8]\d\{1\}|[1-4]\d\{2\}$

Since we only go to n + 1 we stop at i = 3. Therefore,

$R_M = r_3 = [0-8]|5[0-8]\d\{1\}|[1-4]\d\{2\}$

Generate the regular expression for the minimum (R_m).

```
For i = 1 to 2:
     r1 = 4[3-9]
     r2 = [5-9]\d{1}
```

$R_m = r_1 \ \& \ r_2 = 4[3-9]|[5-9]\d\{1\}$

$R = R_M \ \& \ R_m = [0-8]|5[0-8]\d\{1\}|[1-4]\d\{2\}|4[3-9]|$
$[5-9]\d\{1\}$

This can be simplified and finalized to:

$R = ^([0-8]|5[0-8]\d|[1-4]\d\{2\}|4[3-9]|[5-9]\d)$$

A.3 Limitations and Possibilities

The algorithm is limited to non-negative whole numbers in which the data value may be less than or equal to the maximum and greater than or equal to the minimum value. However, regular expressions maybe developed ranges in which the data value may not be equal to the minimum or the maximum. In other words, the range would not have to be inclusive at both ends. In addition, regular expressions may be written for range checking decimal values. Future work would include the enhancement of the algorithm to support these scenarios. Below are two examples of range checking regular expressions involving these scenarios.

For a non-negative number *d* such that $50 < d <= 100$ the regular expression is:

$R = ^(100(?:\.0*)?|[6-9]\d(?:\.\d*)?|5[1-9](?:\.\d*)?|$
$50\.0*[1-9]\d*)$$

For a non-negative number *d* such that $0.0001 <= d <= 1$ the regular expression is:

$R = ^(1(?:\.0*)?|0\.0\{0,3\}[1-9]\d*)$$

APPENDIX B PACRAT Implementation Details

Additional details of the PACRAT implementation discussed above are described in the following subsections.

B.1 Object Oriented Design and Graphical User-Interface

The software design was object-oriented with a graphical user interface designed to be flexible and easy to use. A software requirements document exists and is based on the experience of the customer. The implementation was based on the requirements document and is intended to reduce the number of possible defect injection points and reduce the number of defect escapes in the test application development process. A description of the implementation follows. Below are UML style class diagrams showing the relationships between classes. Modules and forms are also shown in the diagrams to show their relationships as well.

Figure 2 describes the PACRAT software at a high level. When the program runs, subroutine Main in modMain (Main.bas) is called first. After the user logs in, the primary form displayed is a single, global instance of the frmdefCmdsDisplay form. This is the work horse form of the application. Several global objects are used by PACRAT and are included in the Global Objects package displayed in the diagram.

The main form (frmdefCmdsDisplay) of PACRAT contains a menu bar across the top, a status bar across the bottom, and three adjustable width panes in the middle. The left pane contains a tree view of the sections, commands, parameters, and associated test limits in the test application. The middle pane contains tab(s) for editing commands and their parameters and limits, if appropriate. The right pane typically contains tabs for creating and editing predefined setup and limit definitions. The right pane may also contain special forms for editing specific data. Much of the interface for editing data in the test application makes use of grids. The use of grids allows PACRAT to “adjust” itself to the command or parameter being edited.

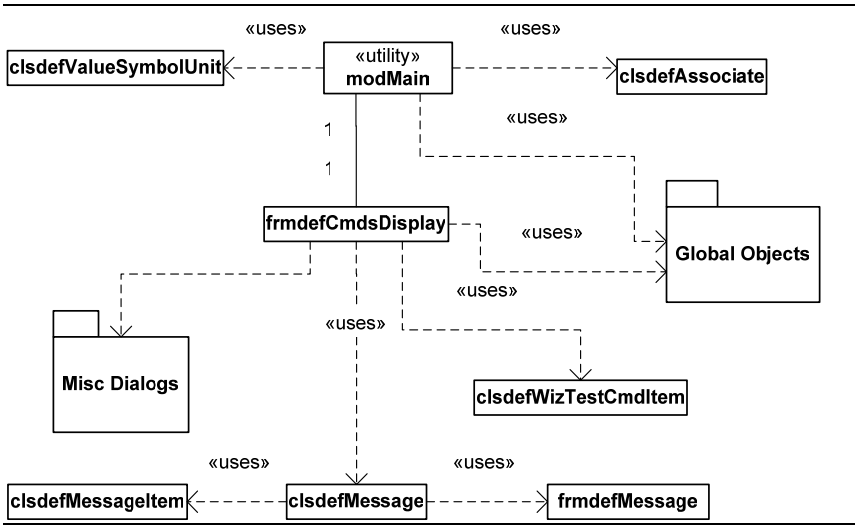

Figure 2 High Level Software Diagram

Miscellaneous dialogs and forms are displayed in Figure 3. In addition, the classes used by these forms are also shown in the diagram. Many of these dialogs and forms also make use of the global objects.

Figure 4 models the global objects used by PACRAT. The hierarchical relationship among some of the classes is evident in this diagram. Many of the classes are collection classes, class wrappers around a member collection object containing objects of an associated class. Since inheritance was not available in the programming language used to develop PACRAT, these wrappers provided a kind of substitute for inheritance. For example, consider the clsdefCommands class which contains a collection of many clsdefCommand objects and provides a means for adding, accessing, and removing these objects. The clsdefCommand class itself contains a clsdefParameters object. Similarly, the clsdefParameters object contains a collection of clsdefParameter objects. Because a test application is primarily made up of commands and their associated parameters, these classes are used extensively throughout the PACRAT application.

The following are descriptions of the global objects used by PACRAT:

g_clsAssociates (clsdefAssociates) – object containing all the associate objects (possible PACRAT users) read in from the Associates.xml file.

g_clsCurrentUser (clsdefAssociate) – object containing the currently logged in PACRAT user.

g_clsCmdDefs (clsdefCommandDefinitions) – object containing all the PT3800 command definition objects read in from the application data database using the Commands query.

g_clsSetupDefs (clsdefSetupDefinitions) – object containing Setup Definition objects read in from the current AM file. (These are user-defined setups that are stored in the test application file.)

g_clsLimitDefs (clsdefLimitDefinitions) - object containing Limit Definition objects read in from the current AM file. (These are user-defined limit definitions stored in the test application file.)

g_clsPatterns (clsdefPatterns) – object containing connector contact patterns read in from the application data database using the ContactPatterns query.

g_clsPreferences (clsdefAppPreferences) – object containing the user preference data read in from the AppPreferences.xml file.

g_clsAM (clsdefTestAM) – object used for reading and writing AM/AP xml files and storing data related to the AM/AP such as the commands, limits, and options.

g_clsLoops (clsdefLoops) – object containing all the test loop objects defined in the command objects in the AM/AP file. Even though the loop objects are stored in the command objects, this collection of all loop objects aids PACRAT in validating rules regarding loops, i.e. there shall be no overlapping or nested loops.

g_clsPowerRanges (clsdefPowerRanges) – object containing all power range objects read in from the application data database using the PowerRanges query.

g_frmCmdsDisplay (frmdefCmdsDisplay) – single instance of the main form that PACRAT uses.

g_clsConnectors (clsdefProdConnectors) – object containing the following collections storing all DefineProdConn commands, all contact names, all contact IDs, all unused contact IDs, all contact names by option bit mask, and unused contact IDs by test option.

g_clsAdapters (clsdefAdapters) – object containing collections of DefineAdapter commands and clsdefAdapterConn objects. This is used in determining which adapters have been used by various types of commands in the AM's Hookup section.

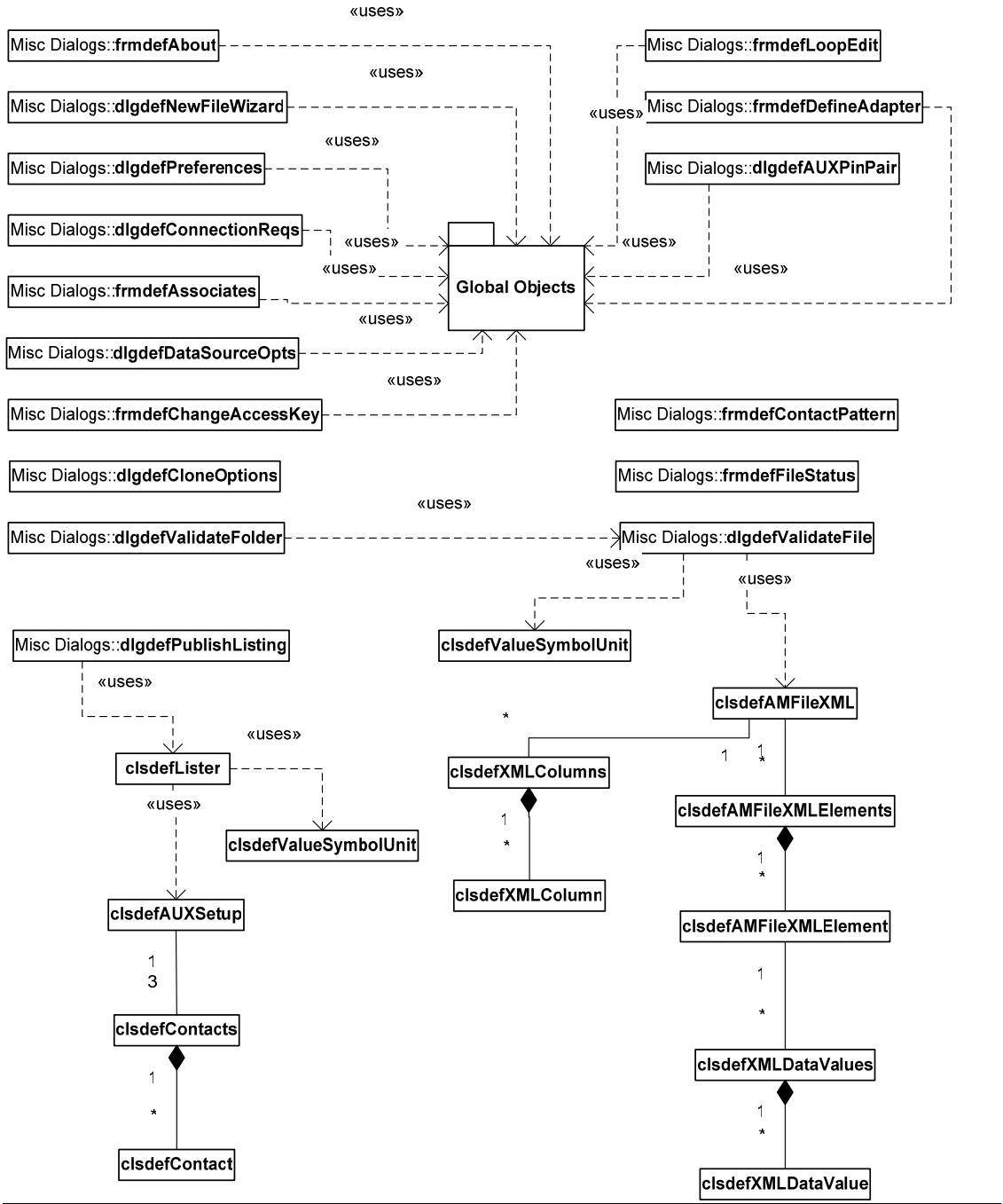

Figure 3 Miscellaneous Dialogs and Forms

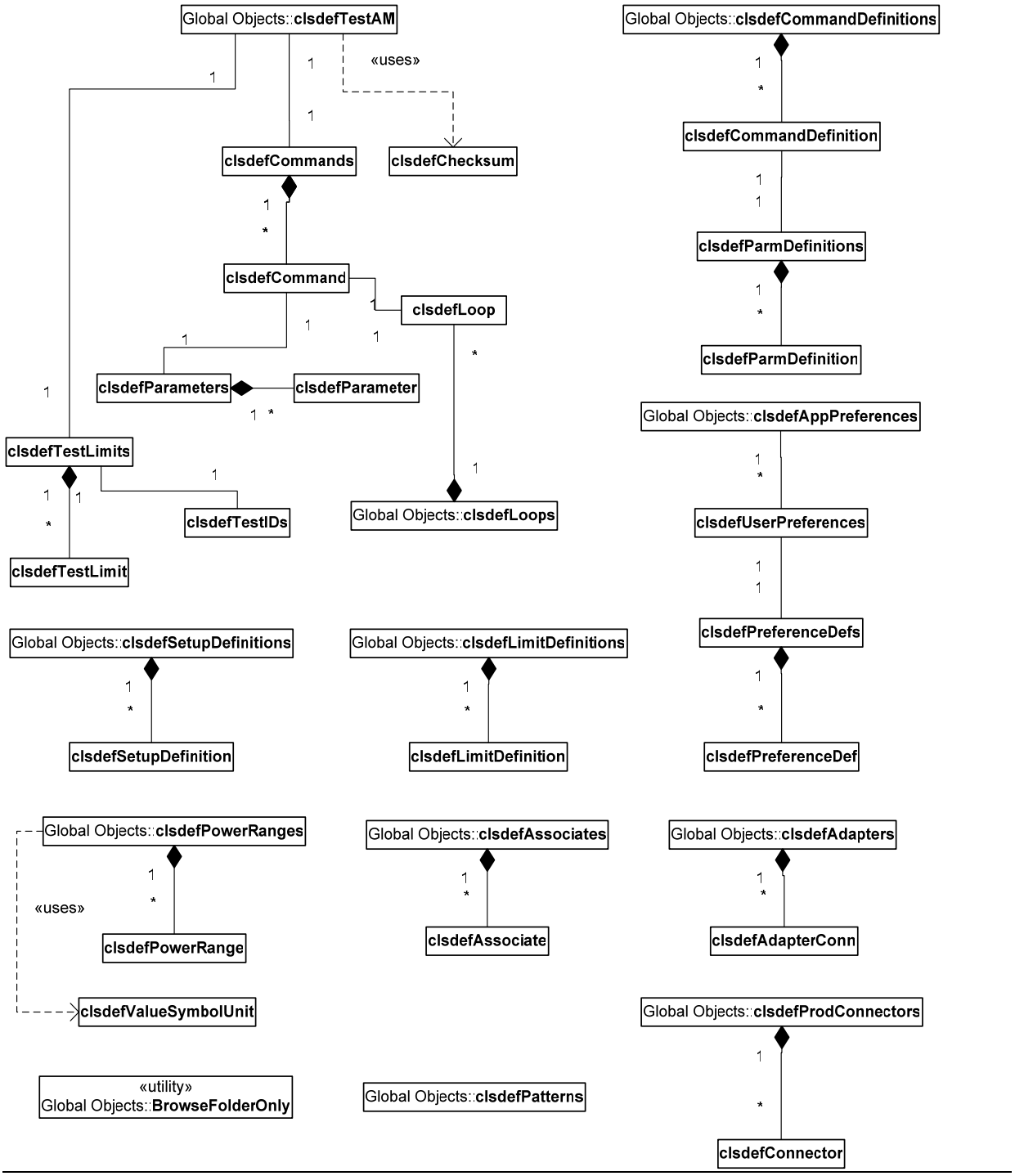

Figure 4 Global Objects and Utilities

B.1.1 Adapter Definition

The define adapter command in PACRAT maps defined product connectors and contacts on those connectors to connectors and contacts on a cable adapter. The

contacts and product connectors used must not be used on another cable adapter and must be defined for all test options enabled for the adapter. It is easy to do this incorrectly. Furthermore, commands that reference adapters, their connectors and/or contacts may only use adapters that are defined for all enabled test options.

In PACRAT, a Define Adapter form was created that allows the user to enter the adapter name, the copy number, and the adapter connector name. These are all verified as unique. A list box displays the available contacts. The “available” contacts are restricted to unused contacts on product connectors that are defined for the test options enabled for the adapter. The user then drags and drops the desired product connector contacts to the desired pin pair on the adapter connector. All the data is validated before it is saved. In addition, the adapter, copy, and connector choices for commands whose parameters refer to defined adapters are restricted to adapters that are defined for all test options enabled for that command.

B.1.2 Product Connector Definition

Test applications for the PT3800 require that product connectors be defined. The parameters include the connector name and a series of contacts. The format of the contacts is `<ContactName>|<ID>|<ScanGroup>`. The ID must be unique. Manually typing this information allows for typos and other errors to be introduced easily. In addition, the product connectors must be defined for all the test options enabled for a given test command that uses the contacts from a given product connector.

PACRAT provides a form to generate a product connector's list of contacts by selecting a pattern and entering the number of contacts and the scan group to use. The list of contacts with unique contact IDs is generated automatically. These may be edited if necessary but the error prone task of manually entering the data is eliminated. In addition, the choices of contacts available for a test command are restricted to contacts from product connectors that are defined for all test options enabled for the test command.

B.2 Database of Application Data

A database was developed defining test application commands and their parameters, the rules regarding their usage, syntax, and appropriate values will be used by PACRAT to aid in defect prevention. The tables contained in this database and their fields are described in the following subsections.

B.2.1 Commands Table

The commands table defines the commands available for the tester and the rules governing their use. The table contains the following fields:

Command – contains an uppercase name of the PT3800 command.
ParmSet – contains the associated parameter set in the Parameters table. A value of -1 indicates that there are no parameters associated with that command.
DisplayFmtdCmd – contains a mixed case version of the Command name above used for display purposes. You should be able to generate the command name by simply converting the **DisplayFmtdCmd** field value to uppercase.
SectionName – contains a regular expression for validating whether or not a given command is allowed in a particular section of the AM file.

CmdLocation – contains a special syntax for defining where a command should go in a given section or sections. A '~' means it may go anywhere in the section. Many commands are allowed in multiple sections and some are required in one or more sections. This field defines those requirements. The syntax is as follows: `$<Section name>:<occurrence such as 1 or 1..*>:<optional command order number>@<optional command that this command must come after>!<optional command that this command must come before>$`. Multiple CmdLocation rules for multiple sections are included on the same line and separated by a '|' character. PACRAT uses a regular expression to parse the elements of the CmdLocation field.

CmdOccurrence – indicates how many occurrences of a given command there should be within an AM file. UML notation is used for indicating ranges (e.g. 1 to many is denoted as "1..*" or 0 to 1 times is shown as "0..1").

Notes – contains special notes regarding a given command. This is useful for the PACRAT software engineer.

CommandType – indicates the type of command (i.e. NORMAL, TEST, TESTANDSETUP, CAL, or CALTEST). A command may be a test command but not use the user-defined setups in PACRAT, thus the distinction between TEST and TESTANDSETUP. A TEST command requires a test ID and limits associated with it. Similarly, a distinction is made between CAL commands (those calibration commands not requiring limits) and CALTEST commands.

Description – a memo field containing a description of what the command does.

DisplayOrder – this field is for sorting purposes. This is the order command definitions are read into the g_clsCmdDefs (clsdefCommandDefinitions) object in PACRAT.

Version – this field indicates the version of the Commands table. Only one command should contain a value in this field. The version number is of the format `<Major>.<Minor>.<Revision>`.

The data in this table is retrieved via the Commands query and is used to populate the global g_clsCmdsDefs (clsdefCommandDefinitions) object.

B.2.2 Parameters Table

This table contains parameter definitions for each parameter set used by the commands. More than one command may reference the same parameter set. The fields contained in this table are:

ParamSet – the parameter set number. This corresponds to the ParmSet field in the Commands table.

ParmGroup – a parameter set may have one or more parameter groups. This field contains the number of the current group.

GroupOrder – this field contains a number indicating the order of the parameters within a given parameter group.

Occurrence – this field indicates how many occurrences of a given parameter there should be within the parameter group. UML notation is used for indicating ranges (e.g. 1 to many is denoted as "1..*" or 0 to 1 times is shown as "0..1").

Parameter – contains the all uppercase parameter name.

DisplayFmtdParameter – contains a mixed case version of the parameter name for display purposes. You should always be able to generate the parameter name by simply converting the DisplayFmtdParameter field to all uppercase.

Validation – contains rules for validating the parameter value. In most cases this is a regular expression. In other cases, keywords are used that are recognized by PACRAT such as $UCONN_CNTCT$. This field is perhaps the most valuable piece of PACRAT.

RuleRegExp – contains a boolean value indicating if the Validation field for this parameter is a regular expression.

Note – contains special notes regarding the parameter for the PACRAT software engineer.

GroupOccurrence - this field indicates how many occurrences of a given parameter group there should be within a given command's parameter set. UML notation is used for indicating ranges (e.g. 1 to many is denoted as "1..*" or 0 to 1 times is shown as "0..1"). However, when a name is used, it identifies the parameter group as a pin specification group.

Default – this field contains an optional default value for the parameter.

UseDefaultForNew – contains a Boolean value indicating if the default should always be used when creating a new command with this parameter. (PACRAT may use other Defaults depending on the value set for another parameter.)

ToolTipText – contains optional text that may be used for a tool tip in PACRAT for the given parameter.

Description – a memo field containing a description of the parameter.

Version – this field indicates the version of the Parameters table. Only one parameter should contain a value in this field. The version number is of the format `<Major>.<Minor>.<Revision>`.

The records from this table are read into an instance of clsdefParmDefs contained in the g_clsCmdDefs object in PACRAT via the Parameters query.

B.2.3 Power Ranges Table

The Power Ranges table contains information regarding valid ranges for voltage and current for certain types of test commands. The fields in this table are:

Command – contains the command name that this range should be used with.

RangeNum – contains the range number for the given command.

VoltageRangeRE – contains a regular expression formally defining a valid range for voltage. This regular expression is used for validation of the range.

VoltRangeDscript – contains a description of the voltage range in more common notation.

CurrentRangeRE – contains a regular expression formally defining a valid range for the current for the given voltage range. This regular expression is used for validation of the range.

CurRangeDscript - contains a description of the current range in more common notation.

Version – this field indicates the version of the Power Ranges table. Only one range should contain a value in this field. The version number is of the format `<Major>.<Minor>.<Revision>`.

The records from this table are read into the g_clsPowerRanges (clsdefPowerRanges) object in PACRAT via the PowerRanges query.

B.2.4 Preference Definitions Table

The user and administrator preference items are defined in the PreferenceDefs table. The table includes the following fields:

TagName – contains the name of a preference item.
TagDisplayName – contains the display name of the preference item.
TagDefault – contains the default value for a given preference item.
TagValidation – contains a regular expression used for validating the preference item value.
AdminAccessRequired – contains a Boolean value indicating if the user must have ADMIN access rights to edit this preference item.
Version – this field indicates the version of the PreferenceDefs table. Only one preference definition item should contain a value in this field. The version number is of the format `<Major>.<Minor>.<Revision>`.

The records from this table are read into an instance of clsdefPreferenceDefs contained in an instance of clsdefUserPreferences contained in the g_clsAppPreferences (clsdefAppPreferences) object in PACRAT via the PreferenceDefs query.

B.2.5 Contact Patterns Table

Connector contact patterns are defined by this table. The fields contained in this table are:

Pattern – this field contains the pattern. Each element in the pattern is separated by a space.
Version – this field indicates the version of the ContactPatterns table. Only one pattern should contain a value in this field. The version number is of the format `<Major>.<Minor>.<Revision>`.

The records from this table are read into the global g_clsPatterns (clsdefPatterns) object in PACRAT via the ContactPatterns query.

B.3 Test Application File Structure

The test application files generated by PACRAT contain the following “sections”:

<dtSoftwareDefs> contains tags and tag values describing the test application itself. There may only be one instance of <dtSoftwareDefs>.
<dtProductDefs> contains tags and tag values describing the product to be tested. There may only be one instance of < dtProductDefs >.
<dtTestAssociations> contains tags and tag values tying a specific test application and its control suffix to a particular product configuration.
<opt> contains information describing a test option. There may be one or more of these entries. Commands in the test application are defined for a specific test option or set of test options.
<id> contains information describing a test ID.

<lm> contains information about acceptance limits used for each test. There is one <lm> entry for every <id> entry.

<sq> contains information describing a command, its parameters, and its associated test ID if applicable. There may be many <sq> entries in a test application.

<image> contains information about an image to be displayed to the tester operator and the actual image data. The image file may be recreated from the image data. In this way, the images are kept with the test application in one file.

<SetupDef> describes user-defined test command setups. These setups allow a test application developer to define a test command setup that may get used repeatedly throughout the test application.

<LimitDef> describes a user-defined limit definition. These limit definitions likewise allow a test application developer to define a limit definition that may be used repeatedly throughout the test application.

B.4 Comprehensive Test Application Validation

PACRAT provides the capability to subject a test application file to a set of comprehensive file validation tests. File validation performs the following:

- Validate Commands
  - Verify that required sections exist
  - Verify that all required commands exist in each section
  - Verify that only allowed commands exist in each section
  - Verify that commands occur in allowed order
  - Verify that command occurrences do not exceed maximum allowed for a given section
- Validate Parameters
  - Verify all required parameters for a given command exist and have a value
  - Verify all required parameters in existing optional groups exist and have a value
  - Verify that pin specification parameters for test commands do not include invalid parameters for the given pin specification
  - Verify that parameter values meet all validation rules specified in the application data database
  - Verify specific parameter values meet requirements for interaction or dependency on other parameters
- Validate Test Command Limits
  - Verify that each test command's base test ID references an existing test limit
  - Verify that the only valid limit types are used
  - Verify that the limit definition is valid for the specified limit type
  - Verify the format of the limit units
  - Verify that the limit units make sense for the type of command
  - Verify valid values for limit pass and fail messages
  - Verify valid value for the limit repeat modifier
- Validate Test IDs
  - Verify the format of the Test IDs
  - Verify that connection verification test IDs begin with "Verify" followed by 1 to 19 digits
  - Verify that other test IDs do not begin with "Verify" and contain no more than 25 characters

    - Verify that each test ID is unique for all enabled options
- Validate Data System IDs
    - Verify that the length of the DataSysID with the repeat modifier does not exceed 6 characters
    - Verify that there is a DataSysID for test commands when the DefineTestOptions Type is “DATASYS”
    - Verify that there is no DataSysID for test commands when DefineTestOptions Type is “OPTIONS”
    - Verify that the DataSysID is unique for all enabled test options
- Validate Test Options
    - Verify that at least 1 test option is defined
    - Verify that no more than 30 test options are defined
    - Verify that the command option bit masks reference defined test options
    - Verify that the command option bit mask is equal to the option bit mask of the first preceding SetOptionsOn command
    - Verify that all defined options are enabled for commands in the Configuration, Setup, TesterChkI, and TesterChkF sections
- Validate Contact IDs
    - Verify that the contact IDs in the DefineProductConn commands are unique
- Validate Defined Adapters
    - Verify that the contact IDs used reference product connectors defined for the enabled options
    - Verify that each defined adapter’s connector name is unique
    - Verify that each defined adapter copy is unique
- Validate Loops
    - Verify there are no overlapping loops
    - Verify there are no nested loops
    - Verify that there is a repeat modifier for each test ID within a loop
    - Verify that the loop starting point and the loop stopping point have the same options enabled
    - Verify that the loop name does not exceed 25 characters
    - Verify that the loop count is >= 2 and <= 1000
    - Verify that the loop pause is >=0 and <=3600 and is a whole number
    - Verify that each test command within a loop has a repeat modifier for the test ID
- Validate Test Command Power
    - Verify that the power-up power for Continuity and ContinuityVerify commands does not exceed 100 Watts
    - Verify that the voltage does not exceed 1000 Volts
    - Verify that the current does not exceed 2 Amps if the voltage is <= 50 Volts
    - Verify that the current does not exceed 1 Amp if the voltage is >50 Volts and <= 100 Volts
    - Verify that the voltage is not negative
    - Verify that the current is not negative
    - Verify AUX current and voltage is within range(s) specified in the application data database
- Validate File Structure
    - Verify that the file is well formed
    - Verify that there is only 1 <dtSoftwareDefs> tag

- Verify that the file status is not ‘ENG’
- Verify that the software drawing number begins with ‘AM’ or ‘AP’
- Verify that the software ID key matches the file name and that it is made up of the drawing number and suffix
- Verify that the drawing number is a development drawing number if the status is ‘DEV’
- Verify that a software load key exists for files with status of ‘ENG’ or ‘DEV’
- Verify there is only 1 <dtProductDefs> tag
- Verify that the number of test IDs in the <id> sections and the number of test IDs in the <lm> sections match
- Verify that there is only 1 <dtTestAssociations> tag